Trabajo Práctico Obligatorio
Diseño de Compiladores
2012

Resumen

El objetivo de este trabajo obligatorio, es la construcción de un intérprete del lenguaje Ruby. Este deberá tomar un programa escrito en Ruby y ejecutar dicho código. A continuación se presenta el lenguaje en cuestión, luego se muestran ciertos aspectos sobre la forma de ejecutar los programas y por ultimo se presentan aspectos mas formales sobre este obligatorio.

Ruby

Ruby es un lenguaje con un balance cuidado. Su creador, Yukihiro “matz” Matsumoto, mezcló partes de sus lenguajes favoritos (Perl, Smalltalk, Eiffel, Ada, y Lisp) para formar un nuevo lenguaje que incorporara tanto la programación funcional como la programación imperativa.

A menudo ha manifestado que está “tratando de hacer que Ruby sea natural, no simple”, de una forma que se asemeje a la vida real. Ruby es simple en apariencia, pero complejo por dentro. En Ruby, todo es un objeto. Se le puede asignar propiedades y acciones a toda información y código.

Ruby es considerado un lenguaje flexible, ya que permite a sus usuarios alterarlo libremente. Las partes esenciales de Ruby pueden ser quitadas o redefinidas a placer. Se puede agregar funcionalidad a partes ya existentes. Ruby intenta no restringir al desarrollador.

En las siguientes subsecciones se presentan los aspectos del lenguaje que deben ser tenidos en cuenta para construir este intérprete. Todo otro aspecto del lenguaje que no este presente en este documento, NO ES REQUERIDO PARA EL INTERPRETE, pero puede ser incorporado se así lo desea. Recuerdo comenzar por lo exigido, y luego en todo caso expandir el intérprete si dispone de tiempo. EL OBJETIVO PRINCIPAL ES ENTREGAR UN INTERPRETE DEL LENGUAJE RUBY QUE FUNCIONE.

Características generales

A continuación se presentan algunos aspectos generales del lenguaje de programación Ruby, así como su entorno de ejecución. Por ejemplo, tenemos que Ruby es/tiene:

- **Libre de formato**: una cosa se puede hacer de distintas maneras. Uno puede escoger el formato que mejor se adapte a nuestra forma de trabajo.
- **Sensible a las mayúsculas y minúsculas**: dos palabras, aunque se diferencien solamente en una letra, por estar en mayúscula o minúscula, son dos cosas distintas. Por ejemplo, 'Dir' no es lo mismo que 'dir'.
- **Múltiples tipos de comentarios**: cualquier línea precedida por '#' es ignorada por el intérprete. Además, cualquier cosa que escribamos entre las líneas '=begin' y '=end,

también será ignorada. (Este tipo de comentarios no pueden tener espacios a su izquierda)

```
# Comentario de una sola línea

=begin
Esto es
un comentario
de varias
líneas
=end
```

- **Delimitadores de instrucción**: varias instrucciones en una misma línea pueden ser separadas por un ';', pero no son necesarios al final de una línea: este final de línea se trata como un ';'. Si un final de línea acaba con un '\', entonces el salto de línea es ignorado, lo que permite tener una instrucción divida en varias líneas.

```
# Varias instrucciones en una misma línea
a=1; b=2; c=3
# es equivalente a:
a = 1
b = 2
c = 3

# Una instrucción en varias líneas
a = "Que bueno que esta interpretar Ruby"
# es equivalente a:
a = "Que bueno que esta \
interpretar Ruby"
```

- **Palabras clave**: Son palabras reservadas, que en Ruby no pueden ser usadas para otros propósitos, por ejemplo, como almacenar valores.
- **'false' y 'nil':** En Ruby, todo esto es válido; de hecho, todo es cierto excepto las palabras reservadas **'false' y 'nil'**.

Ejecutando programas Ruby

Los programas Ruby se almacenan en archivos con extensión "**.rb**", y son procesados por el intérprete de Ruby. Por ejemplo, el clásico Hello World, puede escribirse como:

```
puts 'Hello world'
```

Y lo ejecutamos como:

```
$ ruby hello-world.rb
Hello world
```

También podemos utilizar el Ruby Interactive Mode (IRB), el cual es un interprete que permite ejecutar el código Ruby a medida que tipeemos el mismo.

```
$ irb --simple-prompt
>> 2+2
=> 4
>> 5*5*5
=> 125
>> exit
```

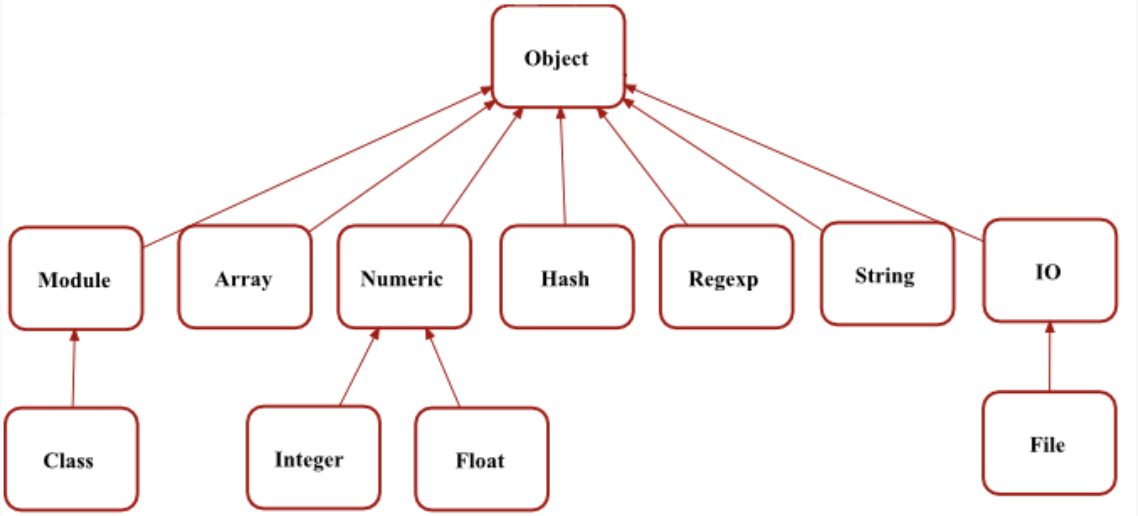
También podemos en ambiente Linux, codificar los programas Ruby con el siguiente formato (ídem scripts PHP):

```
#!/usr/bin/ruby
puts 'Hello world'
```

Si instalamos en ambiente Windows, podemos usar el **Ruby One Click Installer** (descargarlo de http://www.ruby-lang.org/en/downloads), el cual en su proceso de instalación, define las extensiones .rb como ejecutables. Con tipear el nombre del archivo con el programa, este ya será ejecutado

Jerarquía de clases

En el siguiente diagrama se presenta un extracto de la jerarquía de clases utilizada en el lenguaje Ruby



Valores numéricos

En Ruby tenemos valores de tipo entero y de tipo flotante. Por ejemplo:

```
puts 1 + 2
puts 10 - 11
puts 2 * 3
# División entera
puts 3 / 2
# División flotante (al menos un argumento debe ser flotante)
puts 3.0 / 2
puts 3 / 2.0
puts 1.5 / 2.6
```

Operadores

En el siguiente diagrama se presentan los diferentes operadores disponibles en Ruby, así como su precedencia.

:	Alcance (scope)
[]	Índices
**	Exponentes
+ - ! ~	Unarios: pos/neg, no,...
* / %	Multiplicación, División,...
+ -	Suma, Resta,...
> >= < <=	Comparadores
== === <=> != =~ !~	Igualadad, inegualdad,...
&&	'y' booleano
||	'o' booleano
= (+=, -=,...)	Asignadores
?:	Decisión ternaria
not	'no' booleano
and, or	'y', 'o' booleano

Como en otros lenguajes, el uso de paréntesis altera la precedencia. Lo que esta dentro de un grupo de paréntesis, se comporta como un operando, siendo calculado antes en una operación.

Strings

Cualquier cadena de caracteres encerrada entre dos comillas dobles “ o dos comillas simples ‘, es un String. En Ruby, las cadenas son mutables, esto es, pueden ser modificadas.

```
puts "Hola mundo"
# Se puede usar " o ' para los strings
puts 'Hola mundo'
# Juntando cadenas
puts 'Me gusta' + ' Ruby'
# Secuencia de escape
puts 'Ruby\'s party'
# Repetición de strings
puts 'Hola' * 3
# Definiendo una constante
PI = 3.1416
puts PI
```

Hay un tipo especial de string, el delimitado por ` (acento grave). Este tipo de strings, es enviado como comando directamente al sistema operativo. La salida es levantada directamente por Ruby. Por ejemplo:

```
puts `dir`
```

Los strings soportan interpolación. Esto es, la posibilidad de insertar dentro de una cadena, el resultado de una expresión. Para esto, debemos usar la expresión: #{expresión}. Por ejemplo:

```
puts "100 * 5 = #{100 * 5}"
```

Resultaría en una salida: **100 * 5 = 500**

Las interpolaciones y las secuencias de escape, solo funcionan si la cadena esta delimitada por comillas dobles. En caso de ser comillas simples, estas operaciones no se realizan, dejando el string como literal.

Otra propiedad/método interesante de los strings, es la función #length, la cual aplicada a cualquier string, nos da su largo en bytes (generalmente coincide con la cantidad de caracteres)

```
string = "Esto es una cadena"
string.length => 18
```

Variables

Las variables locales en Ruby son palabras que:

- deben empezar con un letra minúscula o un underscore (_)
- deben estar formadas por letras, números y/o underscore (_)

Cuando Ruby encuentra una palabra, la interpreta como: una variable local, un método o una palabra clave. Las palabras claves no pueden ser usadas como variables. Por ejemplo **def** es una palabra clave: sólo se puede usar para definir un método.
Los métodos pueden ser palabras, como **start_here**, **puts** o **print**. Cuando Ruby encuentra una palabra decide qué es de la siguiente forma:

- si hay un signo de igualdad (=) a la derecha de la palabra, es una variable local a la que se le asigna un valor.
- si la palabra se encuentra dentro de la lista de palabras reservadas de Ruby, es una palabra clave.
- Si no se cumple ninguno de los anteriores casos, Ruby asume que es un método.

Asignación dinámica

No hace falta declarar las variables. Al realizar una asignación, se define/redefine la variable según el valor asignado. Por ejemplo:

```
x = 7         #número entero
x = "house"   #string
x = 7.5       #número flotante
```

El método **.class**, aplicado a una variable, devuelve la clase de un objeto.

Alcance

Ruby maneja diferentes alcances para sus variables.

- Alcance global: Las variables globales están precedidas por el signo de $. Pueden ser vistas y usadas en cualquier parte del programa. Ruby maneja ciertas variables globales especiales.
    - $0 es el nombre del archivo que se esta ejecutando
    - $: es el conjunto de directorios en los que Ruby busca un archivo que no existe en el directorio actual
    - $$ contiene el ID de proceso que Ruby esta ejecutando
- Alcance local: Ruby define sus reglas de alcance en base a bloques

Entrada de datos

Puts permite mostrar datos. Gets permite leer datos de la entrada estándar. Por ejemplo:

```
x = gets;
```

Métodos / Funciones

Un método o función es un conjunto de instrucciones que comienza con la palabra **def** y termina con la palabra **end**. Pueden opcionalmente recibir parámetros, como los ejemplos a continuación:

```
def hello
  puts 'Hola'
end

def hello1(nombre)
  puts 'Hola ' + nombre
  return 'correcto'
end
```

Arrays

Algunos ejemplos de arrays:

```
# array vacío
vec1 = []

# Los índices empiezan desde el cero (0,1,2,...)
nombres = ['Satish', 'Talim', 'Ruby', 'Java']
puts nombres[0]
puts nombres[1]
puts nombres[2]
puts nombres[3]

# si el elemento no existe, se devuelve nil
puts nombres[4]

# pero podemos añadir a posteriori más elementos
```

```
nombres[3] = 'Python'
nombres[4] = 'Rails'
```

Los elementos del array pueden ser de tipos diferentes:

```
sabor = 'mango'
vec4 = [80.5, sabor, [true, false]]
puts vec4[2]
```

Los arrays tienen un método especial, denominado length, que indica el largo del array

```
months = Array.new(12)
months.size     # This returns 12
months.length   # This also returns 12
```

Los arrays disponen de muchos métodos utilitarios. Uno de particular interés, es el método **each**. Este puede ser usado para iterar en el contenido del array. Por ejemplo:

```
my_vitamins = ['b-12', 'c', 'riboflavin']

my_vitamins.each do |vitamin|
  puts "#{vitamin} is tasty!"
end

=> b-12 is tasty!
=> c is tasty!
=> riboflavin is tasty!
```

Bloques

Son secuencias de sentencias, encerradas entre llaves **{ }**, o entre **do…end**. Por ejemplo:

Condicionales

En Ruby, **nil** y **false** significan falso, todo lo demás (incluyendo true, 0) significan verdadero. Un ejemplo de if/else/elsif:

```
xyz = 5
if xyz > 4
  puts 'La variable xyz es mayor que 4'
  puts 'Puedo poner más instrucciones dentro del if'
  if xyz == 5
    puts 'Se puede anidar un bloque if,else,end dentro de otro'
  else
    puts "Parte del bloque anidado"
  end
else
  puts 'La variable xyz no es mayor que 5'
  puts 'También puedo poner múltiples sentencias'
end
```

```
if nombre == 'Pedro'
  puts 'Buen nombre!!!'
elsif nombre == 'Pablo'
  puts 'Lindo nombre...'
end
```

También podemos usar la sentencia case:

```
xyz = 10
par = case
  when xyz % 2 == 0 then true
  when xyz % 2 != 0 then false
end
puts par
```

Iteraciones

Disponemos de la sentencia **while** para iterar:

```
var = 0
while var < 10
  puts var.to_s
  var += 1
end
```

Clases y objetos

Ruby es un lenguaje orientado a objetos. Toda entidad que aparece en el lenguaje, es un objeto (por ejemplo, el valor **nil** es considerado un objeto). Las clases se definen de la siguiente forma:

```
# define la clase Perro
class Perro

  # método inicializar clase
  def initialize(raza, nombre)
    # atributos
    @raza = raza
    @nombre = nombre
  end

  # método ladrar
  def ladrar
    puts 'Guau! Guau!'
  end

  # método saludar
  def saludar
    puts "Mi raza es #{@raza} y mi nombre es #{@nombre}"
  end
```

```
end

# para crear  nuevos objetos, se usa el método new
d = Perro.new('Labrador', 'Benzy')

# usamos la instancia ahora
d.ladrar
d.saludar

# con esta variable, apuntamos al mismo objeto
d1 = d
d1.saludar

d = nil
d.saludar
```

Los objetos en Ruby disponen de una serie de métodos utilitarios que simplifican la programación. Por ejemplo:

- object_id: Muestra el identificador único de instancia. `puts "La id del objeto es #{d.object_id}."`
- respond_to?: Permite determinar si un objeto dispone de la implementación del mensaje indicado. Por ejemplo:

```
if d.respond_to?("correr")
  d.correr
else
  puts "Lo siento, el objeto no entiende el mensaje 'correr'"
end
```

- class, instance_of?: El método class devuelve la clase de un objeto dado. El método instance_of? devuelve true si el objeto pertenece a la clase indicada.

```
d = Perro.new('Alsatian', 'Lassie')
puts d.class.to_s # obtenemos Perro
num = 10
puts (num.instance_of? Fixnum) # true
```

Las propiedades de los objetos en Ruby se definen con el simbolo especial @. Estos atributos por defecto no son accesibles desde el exterior. Para acceder a ellos, debemos brindar métodos especiales para leer y/o modificar los datos. Estos métodos se denominan accesores. Por ejemplo:

```
# SIN accesores

class Cancion
  def initialize(titulo, artista)
    @titulo = titulo
    @artista = artista
  end
```

```
  def titulo
    @titulo
  end
  def artista
    @artista
  end
end

cancion = Cancion.new("Brazil", "Olodum")
puts cancion.titulo
puts cancion.artista

# CON accesores

class Cancion
  def initialize(titulo, artista)
    @titulo = titulo
    @artista = artista
  end

  # accesor de lectura
  attr_reader :titulo, :artista

  # accesor de escritura
  # attr_writer :titulo

  # accesor de escritura y lectura
  # attr_accessor :titulo
end

cancion = Cancion.new("Brazil", "Olodum")
puts cancion.titulo
puts cancion.artista
```

Bibliotecas

A medida que los programas crecen, se hace necesario descomponer los mismos en múltiples archivos. Esto significa que para ejecutar un programa, se puede requerir incluir otros archivos aparte del actual. Para eso podemos usar las sentencias load y require.

Por ejemplo, tenemos el archivo **hola.rb** con el contenido

```
puts "Hola a todos!"
```

Ahora hacemos:

```
# busca un archivo con extensión .rb
# si no lo encuentra, sigue buscando algo de nombre 'hola'
require 'hola' # Hola a todos!
```

```
require 'hola' #
require 'hola' #
```

Solo la primera vez que un archivo se requiere, se cargan y/o ejecutan sus instrucciones a memoria. La segunda vez, ya la información esta cargada en memoria.

También podemos hacer:

```
load 'hola.rb' # Hola a todos!
load 'hola.rb' # Hola a todos!
load 'hola.rb' # Hola a todos!
```

La diferencia en este último caso, es que el archivo es cargado cada vez, ejecutando el código si es necesario, en repetidas oportunidades.

Main

En Ruby no tenemos una función especial. El código que se encuentra fuera de funciones o clases, es ejecutado cuando el archivo es cargado.

Puede ser necesario que queramos pasar argumentos al programa Ruby. Por ejemplo,

```
$ ./test.rb test1 test2
```

Podemos acceder a estos argumentos en el programa utilizando la variable predefinida ARGV, la cual es un array que contiene los strings correspondientes a lo pasado como parámetro. Por ejemplo:

```
#!/usr/bin/env ruby

ARGV.each do|a|
  puts "Argument: #{a}"
end

$ ./test.rb test1 test2 "three four"
Argument: test1
Argument: test2
Argument: three four
```

Tratamiento de errores

El requerimiento mínimo es terminar el proceso de traducción una vez que se detecta un error indicando el numero de línea, el ultimo token procesado y de ser posible el tipo de error (sintáctico, de tipo, etc.).

Son aceptables soluciones en los que los mensajes de error son de la forma:

```
Error sintáctico en línea 2 cerca de "22"
```

```
Tipo de datos incorrecto en sentencia IF en línea 10
```

Mecanismos para el manejo de error mas elaborados, serán tenidos en cuenta, pero se sugiere comenzar con un manejo mínimo del error, y luego expandirlo si se tiene tiempo disponible.

Ambiente de ejecución

Deberá implementar un ambiente de ejecución apropiado para permitir ejecutar los programas Ruby que se puedan construir según este obligatorio. Entre otras cosas, este ambiente de ejecución regulara el llamado a funciones, la gestión de memoria, etc.

Ejecución de los programas Ruby

La ejecución de los programas Ruby se realizara en forma similar a como se realiza con el interprete estándar. Esto es, si el nombre del programa desarrollado es INTERPETE, entonces la ejecución será realizada de esta forma:

```
NOMBRE_BINARIO_DE_INTERPRETE archivo.rb arg1 arg2 arg3 ... argN
```

No es necesario proveer un intérprete interactivo como el IRB

Objetivos, formas y plazos de entrega

Algunos aspectos formales a tener en cuenta…

Sobre el objetivo

El objetivo principal del obligatorio es la entrega de un interprete funcional **(TIENE QUE FUNCIONAR)** del lenguaje Ruby, descripto en esta letra. En la sección 2 se presentan las características del lenguaje que deben ser tenidas en cuenta para dicho intérprete. Toda otra característica que no este en este documento, se considera opcional (a menos que sea aclarada su obligatoriedad en la pagina web del curso)

Sobre la forma de trabajo

El trabajo deberá ser realizado en grupos de hasta tres estudiantes. Los mismos deberán ser informados antes del 21/5 al docente encargado del curso. El día 25/5 será informada la lista final de grupos del obligatorio. Se realizaran clases de consulta todos los miércoles hasta la fecha de entrega, en el horario de 8:30 a 10:30, en el salón 301.

IMPORTANTE: Una vez formado un grupo, no se aceptaran separaciones o divisiones en dicho grupo. En caso de darse esta situación, TODOS los integrantes del curso pierden el obligatorio. En caso de que parte de los integrantes abandonen, solo a estos se les considera como no aprobado el curso.

Sobre la plataforma de trabajo

El trabajo deberá ser realizado utilizando el lenguaje C o C++. Se podrán (y se recomienda) utilizar los utilitarios BISON y FLEX para realizar el análisis sintáctico y léxico respectivamente. En caso de ser necesario, se podrá utilizar código descargado de Internet (debidamente

documentado), siempre y cuando este NO RESUELVA una parte fundamental del problema planteado. Por ejemplo, no seria aceptable si se entrega un intérprete desarrollado en C++ de Ruby, descargado de Internet.

Se puede utilizar plataforma Windows o Linux para trabajar. Se recomienda el uso de plataforma Linux (Ubuntu) por su disponibilidad de herramientas. Sin embargo, queda a elección del grupo la plataforma de trabajo

MUY IMPORTANTE:

Se deberá entregar el código fuente de la solución, así como los scripts necesarios para compilar y obtener una versión funcional del intérprete. El grupo se deberá asegurar que el codigo fuente se puede compilar, ya que SOLO se ejecutara el intérprete obtenido de la compilación de los fuentes entregados.

No se aceptaran soluciones que utilicen archivos propietarios de proyectos desarrollados en IDEs como Visual Studio, DJGPP, CodeBlocks, etc. El grupo deberá asegurarse que el código se puede compilar sin necesidad de contar con dichas IDEs

Se recomienda entonces entregar el fuente, junto con un makefile que genere el binario del intérprete.

Sobre la entrega

La fecha de entrega final para este obligatorio es el lunes 2 de julio del 2012 inclusive.

Se deberá entregar un sobre con los datos del grupo, como ser nombres y C.I. de los integrantes. El sobre deberá contener:

- Documentación del obligatorio (breve) presentando los aspectos principales del diseño de la gramática, chequeos, las tablas de símbolos, la representación intermedia, tratamiento de error y el proceso de interpretación. Se espera, muy especialmente, que fundamenten adecuadamente las decisiones de diseño tomadas a lo largo del obligatorio. No se desea un listado del código fuente del intérprete

- Un listado de las pruebas ejecutadas por el grupo sobre el intérprete. En caso de las pruebas que contienen error, indicar los errores obtenidos. En caso de que haya un problema conocido con la ejecución de los programas de prueba entregados, esto deberá ser documentado apropiadamente

- Un cd con los dos puntos anteriores, y el código fuente del interprete

Se aclarara oportunamente en la página web el lugar donde deberá ser entregado este sobre.

Sobre la evaluación

Se evaluaran los siguientes puntos:

- Correcto funcionamiento del intérprete, ejecutando los programas de prueba publicados en la página del curso, y con otros programas de prueba que se consideren necesarios. **SI EL INTERPRETE NO FUNCIONA CORRECTAMENTE, EL CURSO SE CONSIDERA PERDIDO PARA TODOS LOS INTEGRANTES**
- Calidad en la implementación. **SOLO SI EL INTERPRETE FUNCIONA CORRECTAMENTE**, se evaluaran como puntos extra, la calidad con que sea haya implementado la solución, las features extra que se le hayan agregado al intérprete, las mejoras al manejo e informe de errores, etc.
- En caso de duda por parte del docente sobre la implementación del interprete, los integrantes del grupo (en su totalidad o parcialmente) podrán ser llamados a una defensa, en la cual se les realice preguntas concretas sobre la implementación realizada. **EN CASO DE COMPROBARSE QUE UN MIEMBRO DEL EQUIPO NO TRABAJO EN EL OBLIGATORIO, EL CURSO SE CONSIDERA PERDIDO PARA DICHO INTEGRANTE**
- El puntaje máximo del obligatorio es de 100 puntos. Se aprueba con el 60% de los puntos.

Sobre las copias

Para resolver este tema, se utilizara el mecanismo tradicional aplicado en el InCo cuando se detectan copias entre los grupos**. TODOS LOS INVOLUCRADOS PIERDEN EL CURSO** y se informa apropiadamente a la Facultad sobre esta situación.

Referencias

Algunas referencias útiles

- Google: http://www.google.com
- Sitio oficial: http://www.ruby-lang.org/es/
- Tutorial en 20 minutos: http://www.ruby-lang.org/en/documentation/quickstart/
- Tutorial de C: http://www.physics.drexel.edu/courses/Comp_Phys/General/C_basics/